

IB015 Neimperativní programování

Organizace a motivace kurzu,
programovací jazyk Haskell

Jiří Barnat

Organizace kurzu

Cíle kurzu

- Studenti se seznámí s funkcionálním a logickým paradigmatem programování, díky čemuž se odprostí od imperativního způsobu uvažování o problémech a jejich řešení.
- V rámci kurzu se studenti blíže seznámí s funkcionálním programovacím jazykem Haskell a s logickým programovacím systémem Prolog.

Schopnosti absolventa

- Rozumí funkcionálnímu a logickému výpočetnímu paradigmatu.
- Je schopen dekomponovat výpočetní problém na jednotlivé funkce a tuto schopnost používá při vytváření vlastních kódů i v imperativních programovacích jazycích.
- Chápe nedostatky techniky COPY-PASTE programování a umí tuto techniku programování efektivně nahradit.
- Má základní znalost programovacích jazyků Haskell a Prolog.

Předpoklady

- Možné úspěšně absolvovat bez znalosti programování.
- Schopnost abstraktního myšlení.

Znalost imperativního programování

- Je výhodou pro pochopení rozdílného způsobu myšlení v imperativním a neimperativním světě.
- Může být zpočátku mentální bariérou.

Forma ukončení

- Závěrečný písemný test:
 - část I. – test minimálních znalostí (úspěch/neúspěch)
 - část II. – test hlubších znalostí (bodováno 0-35 bodů)
- Možnost získat 15 bodů za domácí úlohy.
- Možnost získat 5 bodů za aktivitu na cvičeních.

Požadavky na úspěšné ukončení

- Úspěšně složit část I. závěrečného testu.
- Získat alespoň 5 bodů za domácí úlohy.

Známkování

- Podle počtu získaných bodů:

E:[5,14] D:[15,24] C:[25,34] B:[35,44] A:[45,55]

Obory

- IB015 je povinná součást studijního základu.
- Doporučen převážně do 3. semestru studia.

Navazující předměty

- IB016 Seminář z funkcionálního programování
- IB013 Logické programování
- IA014 Funkcionální programování
- IA008 Výpočetní logika

Funkcionální paradigma

- <http://haskell.cz/>
- Thompson, Simon. Haskell: the craft of functional programming.
- Structure and Interpretation of Computer Programs
[<http://mitpress.mit.edu/sicp/full-text/book/book.html>]

Logické paradigma

- <http://www.learnprolognow.org>
- Nerode, Shore: Logic for Applications

Co znamená programovat?

Programování

- Vytváření a zápis postupu řešení problému s takovou úrovní detailů a přesnosti, aby tento popis mohl být mechanicky vykonáván strojem, zejména počítačem.
- Zápis postupu = zdrojový kód programu.

Programovací jazyk

- Uměle vytvořený jazyk pro přesný a jednoznačný zápis programů člověkem.

Schopnost programovat

- **Mentální schopnost nacházet mechanicky proveditelné postupy za účelem řešení daného problému.**
- Schopnost přesně formulovat postupy v daném programovacím jazyce.

Volba a znalost programovacího jazyka

- Programovacích jazyků je mnoho.
- Volba programovacího jazyka klade omezení na způsob formulace zamýšlených postupů.

Riziko

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Riziko

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Nedokonalé vs. dokonalé



Riziko

- Dokumentace k programovacím jazykům jsou snadno dostupné i ve formě tutoriálů, avšak samotné poznání syntaxe a sémantiky programovacího jazyka nedělá dokonalého programátora.

Nedokonalé vs. dokonalé



Klasifikace

- Imperativní — C/C++, Java, Perl, php, ...
- Funkcionální – **Haskell**, OCaml, Erlang, Lisp, ...
- Logické – **Prolog**, ...
- Kombinované – C#, Scala, C++, ...
- ...

Jakým jazykem mluví počítač?

- Strojový kód. Program ve strojovém kódu je posloupnost čísel.
- Pro spuštění programu je potřeba provést překlad zdrojového kódu programu do strojového kódu procesoru.
- Překlad se realizuje pomocí **překladače** nebo **interpretru**.
- Pro každý programovací jazyk je potřeba jiný překladač/interpret.

Překladač

- Pro soubor se zdrojovým kódem programu vytvoří soubor obsahující popis programu ve strojovém kódu.
- Výsledný soubor je spustitelný.
- Pracuje se soubory.

Interpret

- Pro daný výraz / příkaz vytvoří odpovídající překlad do strojového kódu a ihned jej provede.
- Nevytváří výsledný spustitelný soubor.
- Často má možnost pracovat interaktivně.
- Pracuje s jednotlivými příkazy/výrazy.

Programovací jazyk Haskell

- Překladač – ghc.
- Interaktivní interpret – ghci (dříve též hugs).
- Neinteraktivní interpretace – runghc.

Překladače programovacího jazyka C/C++

- GNU C++ Compiler (g++, gcc)
- Intel C++ Compiler
- Microsoft Visual C++ Compiler

Programujeme pomocí funkcí

Funkce v programování

- Funkce je předpis jak z nějakého vstupu vytvořit výstup.
- Transformace vstupů na výstupy musí být jednoznačná.

Příklady funkcí

- $f(x) = x*(x+2)$
- objem kvadru $a b c = a*b*c$
- ...

Typ funkce

- Vymezení objektů, se kterými daná funkce pracuje a které vrací na výstup, je součástí definice funkce. Mluvíme o tzv. **typu funkce**.

Příklady

- Funkce, která otočí obrázek o 90 stupňů směrem vpravo.
`rotate90r :: Obrázek -> Obrázek`
- Objem kváдру.
`objemkvadru :: Číslo × Číslo × Číslo -> Číslo`
- Počet hran polygonu.
`hranypolygonu :: Polygon -> Celé_číslo`

Předpoklady

`rotate90r :: Obrázek -> Obrázek`

`hranypolygonu :: Obrázek -> Celé_číslo`

`△ :: Obrázek`

Aplikace funkcí

`rotate90r △` \rightsquigarrow 

`hranypolygonu △` \rightsquigarrow 3

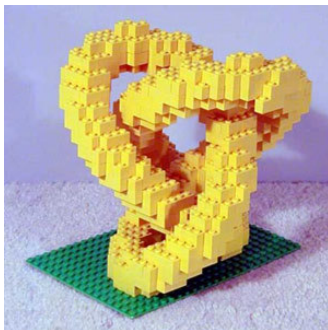
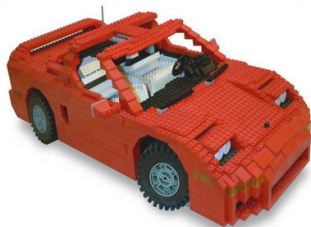
Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Pozorování

- Složitější úkony lze realizovat pomocí jednodušších operací.
- Složitější funkce lze definovat **složením** jednodušších.

Skládání – cesta ke složitějším objektům a funkcím



Operátor .

- $(f1 . f2) x = f1 (f2 x)$
- Čteme jako „f1 po f2“.

Příklad

- Předpokládejme funkci `double`, která vezme obrázek a vytvoří nový zkopírováním vloženého obrázku dvakrát vedle sebe.



`double :: Obrázek -> Obrázek`

`double`  \rightsquigarrow 

- Novou funkci `rotate_and_double` můžeme definovat takto:

`rotate_and_double :: Obrázek -> Obrázek`

`rotate_and_double x = (double . rotate90r) x`

`rotate_and_double`  \rightsquigarrow 

Složené funkce a η -redukce

- Složení funkcí je možné definovat bez uvedení parametru.
- Tj. definici

```
rotate_and_double x = (double.rotate90r) x
```

lze zapsat také jako


```
rotate_and_double = double.rotate90r
```


POZOR na prioritu vyhodnocování v Haskellu


- Aplikace funkce na parametry má nejvyšší prioritu.



```
double.rotate90r  $\triangle$  = double.(rotate90r  $\triangle$ )  $\rightsquigarrow$  ERROR
```


- Závorky kolem výrazu `double.rotate90r` jsou při aplikaci na hodnotu \triangle nutné.


`(rotate_and_double . rotate_and_double)`  \rightsquigarrow



`((double . double) . double)`  \rightsquigarrow

`(double . hranypolygonu)`  \rightsquigarrow


`(rotate_and_double . rotate_and_double)`  \rightsquigarrow 



`((double . double) . double)`  \rightsquigarrow

`(double . hranypolygonu)`  \rightsquigarrow


`(rotate_and_double . rotate_and_double)`  \rightsquigarrow 

`((double . double) . double)`  \rightsquigarrow 

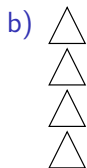
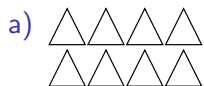
`(double . hranypolygonu)`  \rightsquigarrow

`(rotate_and_double . rotate_and_double)`  \rightsquigarrow 

`((double . double) . double)`  \rightsquigarrow 

`(double . hranypolygonu)`  \rightsquigarrow **ERROR**

Jak lze pomocí `double`, `rotate90r` a \triangle vyrobit následující?



Typová signatura:

`rotate_and_double :: Obrázek ->Obrázek`

Jméno funkce

`rotate_and_double` x = (double.rotate) x

Tělo funkce

`rotate_and_double` x = (double.rotate) x

Definice funkce

`rotate_and_double` x = (double.rotate) x

Formální parametr

rotate_and_double x = (double.rotate) x

Aktuální parametr

rotate_and_double △

Výraz

rotate_and_double △

Podvýraz

rotate_and_double (rotate_and_double △)

Funkcionální programování v Haskellu

Funkcionální výpočetní paradigma

- program = výraz + definice funkcí
- výpočet = úprava (zjednodušení) výrazu
- výsledek = hodnota (nezjednodušitelný tvar výrazu)

Příklad programu

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Program

- definice funkcí

```
square x = x * x
```

```
pyth a b = square a + square b
```

- výraz

```
pyth 3 4
```

Výpočet

$$\begin{aligned} \underline{\text{pyth } 3 \ 4} &\rightsquigarrow \underline{\text{square } 3} + \text{square } 4 \rightsquigarrow 3 * 3 + \underline{\text{square } 4} \rightsquigarrow \\ &\rightsquigarrow \underline{3 * 3} + 4 * 4 \rightsquigarrow 9 + \underline{4 * 4} \rightsquigarrow \underline{9 + 16} \rightsquigarrow \\ &\rightsquigarrow 25 \end{aligned}$$

Lokální definice

- Definují symboly (funkce, konstanty) pro použití v jednom výrazu, vně tohoto výrazu jsou tyto symboly nedefinované.
- Lokální definice mají vyšší prioritu než globální definice.

V Haskellu pomocí let ... in

- let definice in výraz

```
let fcube x = x * x * x in fcube 12
```

```
let fcube x = x * x * x in let c = 12 in fcube c
```

```
let fcube x = x * x * x; c = 12 in fcube c
```

Čísla

- `Integer` – libovolně velká celá čísla
- `Int` – celá čísla do velikosti slova procesoru
- `Float` – reálná čísla
- `Fractional` – racionální čísla

Znaky a řetězce

- `Char` – znak, příklady hodnot: `'a'`, `'2'`, `'>'`
- `String` – řetězec, například: `"Toto je řetězec."`
- `String` je totéž co `[Char]`

Pravdivostní hodnoty

- `Bool`
- Typ `Bool` má pouze 2 hodnoty: `True` a `False`

Příklad

- Definujte funkci `jedna_nebo_dva`, která vrátí `True` pokud dostane na vstupu číslo 1 nebo 2, jinak vrátí `False`.

```
jedna_nebo_dva :: Integer -> Bool
jedna_nebo_dva 1 = True
jedna_nebo_dva 2 = True
jedna_nebo_dva _ = False
```

Víceřádkové definice funkcí

- Na místě formálních parametrů se použijí tzv. vzory.
- Použije se první vzor, který vyhovuje.
- Symbol `_` vyhovuje libovolnému parametru.
- Lze použít pro větvení výpočtu.

Podmíněný výraz

- if *podmínka* then *výraz1* else *výraz2*
- *podmínka* – výraz, který se vyhodnotí na hodnotu typu `Bool`
- *výraz1* se vyhodnotí pokud se podmínka vyhodnotí na hodnotu `True`, *výraz2* se vyhodnotí, pokud se podmínka vyhodnotí na hodnotu `False`.
- Výrazy *výraz1* a *výraz2* musejí být stejného typu.

Test na rovnost

- Pro dotaz na rovnost používáme symbol `==`.
- `3 == 4` \rightsquigarrow `False`
- `3 = 4` \rightsquigarrow **Error**

Možnosti zápisu binárních funkcí

- Infixový zápis binárních funkcí: $3+4$, $4*5$
- Prefixový zápis binárních funkcí: $(+) 3 4$, $(*) 4 5$

Volání funkce a parametry

- Jméno funkce a použité parametry jsou odděleny mezerou, pokud je některý z parametrů výraz, který je sám o sobě aplikace funkce na argumenty, je třeba celý tento výraz ozávkovat.
- $(*) 3 4 + 5 \rightsquigarrow 17$
- $(*) 3 + 4 5 \rightsquigarrow$ **Error**
- $(*) 3 (+) 4 5 \rightsquigarrow$ **Error**
- $(*) 3 ((+) 4 5) \rightsquigarrow 27$

Vstup výstupní operace

- Operace, které čtou a zapisují data z/do souborů, či terminálu.
- Program v Haskellu je reprezentován výrazem `main`.
- Program je sekvence vstup-výstupních akcí.
- Funkcionální princip vstup/výstupních akcí je složitý, bude vysvětlen později.

do notace programu v Haskellu

- ```
main = do putStr "Zadej celé číslo: "
 x <- getLine
 print ((+) 1 (read x :: Int))
```
- Textové zarovnání je důležité!

## Zadání

- Napište a spusťte program v Haskellu, který bude řešit dělitelnost dvou čísel, tj. zeptá se uživatele na dělence, načte ho, pak se zeptá na dělitele, kterého také načte, a sdělí uživateli, zda je zadaný dělenec dělitelný beze zbytku zadaným dělitelem.